УДК 510.643
DOI: 10.25559/SITITO.16.202003.543-550

Original article

Dynamic Separation Logic and its Use in Education

E. M. Makarov
Lobachevsky State University of Nizhny Novgorod, Nizhny Novgorod, Russian Federation
23 Gagarin Av., Nizhny Novgorod 603950, Russian Federation
evgeny.makarov@itmm.unn.ru

Abstract

Mathematical logic is widely used in hardware and software verification. Hoare logic is particularly suitable for reasoning about imperative programs. Its extension, separation logic, introduces the separating conjunction, which makes it possible to reason about programs working with pointers and mutable data structures. Dynamic logic, an example of modal logic, is yet another formalism used for verification. This article introduces propositional dynamic separation logic, which adds separating conjunction to dynamic logic.

We describe syntax, semantics and Hilbert-style deductive system for propositional dynamic separation logic and prove its soundness. The definition of the logic is rather abstract. Thus, the programming language consists of so-called regular programs rather than while-programs, and the set of atomic commands can be arbitrary as long as they correspond to local actions. Special attention is devoted to the soundness of the frame rule, which allows writing program specification using a small footprint, i.e., specifying exactly the portion of the heap that the program reads or writes. Programs that perform tests are also treated differently from regular dynamic logic.

The article also argues for the use of separation logic in computer science curriculum. It is more intuitive that other substructural logics and can be taught even in introductory logic courses. At the same time, it is an active research area with numerous verification tools built on its foundation. Therefore, it serves an excellent introduction to formal methods.

Keywords: dynamic separation logic, frame rule, soundness.

The author declares no conflicts of interest.

For citation: Makarov E.M. Dynamic Separation Logic and its Use in Education. *Sovremennye informacionnye tehnologii i IT-obrazovanie* = Modern Information Technologies and IT-Education. 2020; 16(3):543-550. DOI: https://doi.org/10.25559/SITITO.16.202003.543-550

Оригинальная статья

Динамическая логика разделений и ее использование в образовании

Е. М. Макаров
ФГАОУ ВО «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского», г. Нижний Новгород, Российская Федерация
603022, Российская Федерация, г. Нижний Новгород, пр. Гагарина, д. 23
evgeny.makarov@itmm.unn.ru

Аннотация
Математическая логика широко применяется для верификации программ и аппаратного обеспечения. Одной из логик, наиболее подходящих для верификации императивных программ, является логика Хоара. Ее расширение, логика разделений, использует разделяющую конъюнкцию, которая позволяет рассуждать о программах, использующих указатели и изменяемые структуры данных. Еще одним формализмом, используемым в верификации, является динамическая логика — частный случай модальной логики. Данная статья описывает пропозициональную динамическую логику разделений, которая добавляет разделяющую конъюнкцию к динамической логике.
Мы описываем синтаксис, семантику и исчисление в гильбертовском стиле для пропозициональной динамической логики разделений и доказываем ее корректность. Определение логики является достаточно абстрактным. Так, используемый язык программирования состоит из так называемых регулярных программ вместо программ с обычными конструкциями if и while. Множество атомарных программ может быть произвольным при условии, что они порождают локальные действия. Особое внимание уделяется корректности правила кадра, позволяющего писать локальные спецификации программ, то есть указывать участки памяти, которые непосредственно читаются или изменяются программой. Условные операторы также рассматриваются отлично от стандартной динамической логики.
Статья также содержит аргументы в пользу использования логики разделений в изучении компьютерных наук. Данная логика более проста, чем другие субструктурные логики, и может преподаваться даже в начальных курсах математический логики. В то же время, логика разделений является областью активных исследований с большим количеством применений. На основе этой логики построено много инструментов для верификации. Поэтому она является отличным введением в формальные методы.

Ключевые слова: динамическая логика разделений, правило кадра, корректность.

Автор заявляет об отсутствии конфликта интересов.

Для цитирования: Макаров, Е. М. Динамическая логика разделений и ее использование в образовании / Е. М. Макаров. — DOI 10.25559/SITITO.16.202003.543-550 // Современные информационные технологии и ИТ-образование. — 2020. — Т. 16, № 3. — С. 543-550.

Introduction

Mathematical logic has long been used to help create and verify computer software [1]. While generic first- and higher-order logics can be used to prove properties of programs,[1] in practice specialized logics are necessary. One of the oldest examples of such logic is Hoare logic [2; 3]. It uses formulas, or assertions, of first-order logic (or some other chosen logic) to describe contents of variables, or states. The derivable judgments of Hoare logic, called triples, have the form $\{F\}C\{G\}$, where F and G are assertions, called pre- and postconditions, respectively, and C is a program, or command. Such triple may express partial or total correctness of the program C. In the first case, the triple means that if C starts in a state satisfying F and finishes execution, then the final state satisfies G. Total correctness additionally demands that C terminates.

As a calculus, Hoare logic provides axiomatic semantics to programs. It includes a collection of axioms and inference rules that allow deriving specifications of programs.

A serious shortcoming of Hoare logic has to do with shared mutable data structures. When a program is allowed to use pointers, a single field can be referenced from different places, and updating the content of one pointer may unexpectedly change the content of a seemingly unrelated pointer. This situation is known as aliasing. To deal with such programs, in early 2000s Reynolds, O'Hearn and others invented separation logic [4; 5].

Assertions and reasoning about them in separation logic is borrowed from the so-called logic of bunched implications (BI logic) [6]. During the early days of separation logic assertions were primarily though of as predicates on heaps, i.e., memory segments. BI logic adds to classical logic new connectives $*$ and $-\!\!*$, called separating conjunction and separating implication. Assertion $F_1 * F_2$ is considered true on a heap σ if it can be divided into two disjoint portions σ_1 and σ_2 such that F_1 is true on σ_1 and F_2 is true on σ_2. The connective $-\!\!*$ is an implication that approximately relates to $*$ and ordinary implication relates to conjunction. To talk about heaps, separation logic uses the predicate $l \mapsto v$, which means that value v is stored in the heap at location l. Various axioms, for example, saying that the $\mapsto$ is a partial function, can be added. Separation logic also includes an important *frame rule*.

$$\frac{\{F\}C\{G\}}{\{F * H\}C\{G * H\}} \tag{1}$$

It means that if C converts a heap satisfying F into one satisfying G, then it will work similarly when run on a larger heap and won't touch the excess part. This rule allows writing function specification using small footprint, i.e., specifying exactly the portion of the heap that the function reads or writes. Using the frame rule, this specification can be lifted to reason about a larger program.

Hoare logic and separation logic have enjoyed great success as machinery for specifying and proving program correctness. Many software tools were created for verifying software with different degrees of human participation. Some successful projects are listed in Section 4.

Dynamic logic [7] is yet another formalism for reasoning about programs. Unlike Hoare and separation logics, its judgments are not restricted to triples and can be arbitrary formulas. It is a modal logic that for every program C includes a modal operator $[C]$. Formula $[C]F$ is true in a state σ if every terminating execution of C starting in σ ends in a state satisfying F. Thus, Hoare triple $\{F\}C\{G\}$ can be expressed as $F \to [C]G$. As a result, dynamic logic is at least as expressive as Hoare logic.

Dynamic logic has also been quite useful for practical verification. For example, the KeY Project [8] uses dynamic logic in a tool that allows verifying Java programs against specifications written in the Java Modeling Language.

Since Hoare logic can be embedded into dynamic logic, it makes sense to extend the latter by adding separating connectives. To our knowledge, this has not been done. So the first contribution of this paper is a formulation of propositional dynamic separation logic and a proof of its soundness. The second contribution is a discussion of why dynamic and separation logics form a particularly suitable subject in computer science education.

This paper deals with propositional dynamic separation logic and, following [9], abstracts from many concrete details of a programming language. For this reason it does not show examples of verifying realistic algorithms. Several examples, including in-place reversal of a linked list, copying a tree and Schorr-Waite graph marking algorithm, can be found in [4].

The outline of this paper is as follows. Section 2 introduces dynamic separation logic and describes its similarity with traditional separation logic. Section 3 proves soundness of dynamic separation logic. The use of dynamic and separation logics in education is discussed in Section 4. Finally, Section 5 concludes and points to future research.

Dynamic Separation Logic

This section introduces dynamic separation logic. Most of the technical details are based on [9] and are similar to regular separation logic. The main difference is that separation logic uses triples while dynamic separation logic uses formulas of the BI logic with added modal operators $[C]$.

Definition 1. Formulas F and commands C are described using the following mutually recursive grammar.

$$F, G ::= p \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{emp} \mid \neg F \mid F \wedge F \mid F \vee F \mid F \to F \mid F * F \mid [C]F$$
$$C ::= a \mid C; C \mid C + C \mid C^* \mid F?$$

The command $C_1; C_2$ executes C_1 followed by C_2; $C_1 + C_2$ nondeterministically chooses and runs one of C_1, C_2; C^* executes C zero or more times, while $F?$ does nothing if F is true in the current state and terminates the process normally otherwise (instead of terminating the process may be though of as diverging since we are interested in partial correctness). This language is nondeterministic due to the presence of $+$, so there may be multiple computations starting in the same state.

In this programming language some operators can be expressed using well-known translation [7, Sect. 5.1].

$$\mathsf{skip} = \mathsf{true}?$$
$$\mathsf{fail} = \mathsf{false}?$$
$$\mathsf{if}\ F\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 = (F?; C_1) + (\neg F; C_2)$$
$$\mathsf{while}\ F\ \mathsf{do}\ C = (F?; C)^*; \neg F?$$
$$\mathsf{repeat}\ C\ \mathsf{until}\ F = C; (\neg F?; C)^*; F?$$

Variable p ranges over the countable set of propositional variables, and a ranges over the set of atomic commands. The following example is taken from [5].

[1] For example, it is well-known that Kleene's *T* predicate, which describes the computation of a given Turing machine, is representable in formal arithmetic and so can be used to express and formally prove properties of algorithms encoded by Turing machines.

Example 2. Let arithmetic expressions E and atomic actions a be defined by the following grammar.

$$E ::= x, y, \ldots \mid 0 \mid 1 \mid E + E \mid E \times E \mid E - E$$
$$a ::= x := E \mid x := [E] \mid [E] := E \mid x := \mathsf{cons}(E, \ldots, E) \mid \mathsf{dispose}(E)$$

Here $[E]$ denotes the content of memory, or heap, at address E. Therefore, $x := [E]$ reads the content of memory at address E into x; $[E_1] := E_2$ writes the value of E_2 to address E_1; $x := \mathsf{cons}(E_1, \ldots, E_n)$ allocates n consecutive cells, writes values of $E_1, \ldots, E_n$ there and assigns x to the address of the first of those cells; and $\mathsf{dispose}(E)$ frees the memory cell at address E. Commands that access the heap may generate a fault if the required address is not allocated.

Commands are interpreted as binary relations on the set Σ of *states*. Historically the first example of states was heaps possibly paired with *stores*, i.e., values of local variables. A crucial feature of heaps is that they support the operation of union of disjoint portions of a heap. Over time, many other interpretations of states were proposed. The following definition generalizes over them.

Definition 3. A separation algebra is a triple $(\Sigma, *, e)$ where $*$ is a partial binary operation on Σ and $e \in \Sigma$ satisfying the following properties for all $\sigma_1, \sigma_2 \in \Sigma$.
1. $\sigma_1 * \sigma_2 = \sigma_2 * \sigma_1$;
2. $e * \sigma = \sigma * e = \sigma$;
3. $\sigma_1 * \sigma = \sigma_2 * \sigma \implies \sigma_1 = \sigma_2$.

Thus, Σ is a partial commutative monoid. We write $\sigma_1 \preceq \sigma_2$ if $\sigma_2 = \sigma_1 * \sigma$ for some σ and $\sigma_1 \# \sigma_2$ if $\sigma_1 * \sigma_2$ is defined.

Example 4. Let $\Sigma = \mathsf{Heaps} = L \rightharpoonup_{\mathrm{fin}} V$ where L is a set of locations, V is a set of values, and $\rightharpoonup_{\mathrm{fin}}$ denotes the set of finite partial functions. Then e denotes the nowhere defined function and $*$ is the union of functions with disjoint domains.
One can take $L = \mathbb{N}$ and $V = \mathbb{Z}$ [5]. For another example, consider unspecified disjoint sets Atoms and $\mathsf{Addresses}$, let $L = \mathsf{Addresses}$ and V be the set of all finite tuples whose elements come from $\mathbb{Z} \cup \mathsf{Atoms} \cup \mathsf{Addresses}$ [4]. This construction allows representing linked lists, i.e., sequences of pairs whose second element is either an atom nil or an address of the following pair.

Example 5. Let Var be the set of local variables and L and V be as in the previous example. Then $\Sigma = \mathsf{Stores} \times \mathsf{Heaps}$ where $\mathsf{Stores} = \mathsf{Var} \rightharpoonup_{\mathrm{fin}} V$. This represents the variables-as-resource model [10].

Example 6. Let $V = \mathbb{Z} \times (0, 1]$. Then heaps with values from V can be viewed as collections of cells with permissions. Permission 1 gives the process full access to the cell and allows reading, writing and freeing it. Permissions strictly smaller than 1 give only reading access. The union of heaps is defined as follows.

$$(h_1 * h_2)(l) = \begin{cases} (v, p), & \text{if } h_1(l) = (v, p),\ h_2(l) \text{ is undefined} \\ (v, p), & \text{if } h_2(l) = (v, p),\ h_1(l) \text{ is undefined} \\ (v, p_1 + p_2), & \text{if } h_1(l) = (v, p_1), h_2(l) = (v, p_2), p_1 + p_2 \le 1 \end{cases}$$

This definition allows sharing permissions between several processes.

Definition 7. A predicate on a separation algebra $(\Sigma, *, e)$ is a subset of Σ. Variables P, Q, R range over predicates.

As usual in dynamic logic, the semantics of formulas is defined by a function $[\![\cdot]\!]$ mapping formulas to predicates. We write $\sigma \models F$ if $\sigma \in [\![F]\!]$. Similarly, the semantics of commands is usually described by a function that we'll also denote $[\![\cdot]\!]$ mapping commands to binary relations on Σ. This turns Σ into a Kripke frame [7, Sect. 3.7] and leads to the standard definition

$$\sigma \models [C]F \stackrel{\text{def}}{\iff} \forall \sigma'.\, \sigma [\![C]\!] \sigma' \implies \sigma' \models F.$$

However, one has to decide how to incorporate memory faults into this definition. A computation starting from a certain state may behave in one of the following ways:
1. finish normally and produce a final state;
2. terminate (for example, after executing $F?$) without producing a final state;
3. run forever;
4. terminate with a memory fault.

Options 2 and 3 do not violate partial correctness, but 4 does. Namely, if one computation of C starting in σ produces a fault, then $\sigma \not\models [C]F$ by definition, even if other computations of C finish successfully. Thus we need to distinguish between cases 2 and 4 above.
In the early days of separation logic [5] this was done by defining a state σ *safe* for C if option 4 above never happens when C is run in σ. This concept was then used to formulate the soundness of the frame rule. However, [9] introduced a more compact way to define $\sigma \models [C]F$ that automatically takes memory faults into account. Namely, instead of a binary relation $[\![C]\!]$ is viewed as a function from Σ to $P(\Sigma)$ where $P(\cdot)$ denotes a powerset. Then $\sigma \models [C]F$ holds if $[\![C]\!](\sigma) \subseteq [\![F]\!]$. To deal with faults, we give the following definition.

Definition 8. Let $P(\Sigma)^\top \stackrel{\text{def}}{=} P(\Sigma) \cup \{\top\}$ where $\top \notin P(\Sigma)$ is a new greatest element. The order $\sqsubseteq$ is defined as follows: $U \sqsubseteq V$ holds iff $V = \top$ or $U, V \in P(S)$ and $U \subseteq V$. The set $P(\Sigma)^\top$ is clearly a complete lattice, whose join and meet are denoted by $\sqcup$ and $\sqcap$. Variables U, V range over $P(\Sigma)^\top$.
The greatest element $\top$ of $P(\Sigma)^\top$ should not be confused with $[\![\mathsf{true}]\!] = P(\Sigma)$. In particular, $\top$ is not a predicate and $\sigma \models \top$ is not defined.

Definition 9. An action is a function from Σ to $P(\Sigma)^\top$. The set of all actions is a complete lattice under pointwise order: $f \sqsubseteq g$ if $f(\sigma) \sqsubseteq g(\sigma)$ for all $\sigma \in \Sigma$. Somewhat abusing the notation, the join and meet on the set of actions is also denoted by $\sqcup$ and $\sqcap$. Variables f, g range over actions.
A semantics maps commands to actions. It has to be defined in such a way that if C *may* arrive at a memory fault starting in σ, then $[\![C]\!](\sigma) = \top$. Then the definition

$$\sigma \models [C]F \quad \text{holds if} \quad [\![C]\!](\sigma) \sqsubseteq [\![F]\!]$$

has the desired effect: it implies that $\sigma \not\models [C]F$ if C produces a fault because $[\![F]\!] \in P(\Sigma)$ and therefore $\top \sqsubseteq [\![F]\!]$ is false.
It will be shown in Section 3 that the following definitions characterizes actions satisfying the frame rule (1).

Definition 10. An action $f \colon \Sigma \to P(\Sigma)^\top$ is called local if $f(\sigma_1 * \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\}$ for all $\sigma_1 \# \sigma_2$.
If f is local, then $f(\sigma_1) \neq \top$ implies that $f(\sigma_1 * \sigma_2) \neq \top$. For $f = [\![C]\!]$ this means that if C does not produce a memory fault starting in σ_1, than it won't produce a fault starting in a larger

state $\sigma_1 * \sigma_2$. Further, if C starts in $\sigma_1 * \sigma_2$, then every final state can be split into a σ_1' and σ_2 where C converts σ_1 into σ_1'. Thus, C does not touch the additional portion σ_2 of the state, which is exactly what the frame rule says.

It may seem at first that the condition on actions that is equivalent to the frame rule can be states as follows.

$$\sigma_1' \in f(\sigma_1) \implies \sigma_1' * \sigma_2 \in f(\sigma_1 * \sigma_2)$$

But the allocation operator from Example 2 does not satisfy this property. Indeed, consider the separation algebra consisting of heaps from Example 4. Let $[l_1 \mapsto v_1, \dots, l_n \mapsto v_n]$ denote the heap that maps l_i to v_i (all l_i are assumed to be different). Then $x := \mathsf{cons}(1)$ may convert the empty heap e to $[0 \mapsto 1]$. But if $x := \mathsf{cons}(1)$ is executed in a heap where 0 is already allocated, for example, in $[0 \mapsto 0]$, then it has to allocate a new address, so $x := \mathsf{cons}(1)$ cannot convert $e * [0 \mapsto 0]$ to $[0 \mapsto 1] * [0 \mapsto 0]$. Note, however, that $x := \mathsf{cons}(1)$ may convert $e * [0 \mapsto 0]$ to, say, $[1 \mapsto 1] * [0 \mapsto 0]$, and the same instructions is allowed to change e into $[1 \mapsto 1]$ in accordance with the definition of local action.

It is proved in [5] that atomic actions from Example 2 are local. One example of a non-local action is a constant function $f(\sigma) = \{e\}$. Indeed, Definition 10 requires that $f(\sigma_1 * \sigma_2) = \{e\} \subseteq f(\sigma_1) * \{\sigma_2\} = \{e\} * \{\sigma_2\} = \{\sigma_2\}$, which is false in general. Another non-local action g assigns the same value, say, 0, to all allocated cells. Then g converts $[0 \mapsto 1]$ to $[0 \mapsto 0]$, but g does converts $[0 \mapsto 1, 1 \mapsto 2]$ to $[0 \mapsto 0, 1 \mapsto 0]$ rather than $[0 \mapsto 0, 1 \mapsto 2]$, as expected of a local action.

Now we are ready to define the semantics of formulas and commands.

Definition 11. A model is a pair of separation algebra $(\Sigma, *, e)$ and a function $[\![\cdot]\!]$ mapping propositional variables to predicates and atomic commands to local actions. We define the following operations on elements of $P(\Sigma)^\top$ (recall that $P, Q \in P(\Sigma)$ and $U \in P(\Sigma)^\top$).

$$\neg P \stackrel{\text{def}}{=} \Sigma \setminus P$$
$$P \wedge Q \stackrel{\text{def}}{=} P \sqcap Q$$
$$P \vee Q \stackrel{\text{def}}{=} P \sqcup Q$$
$$P \to Q \stackrel{\text{def}}{=} (\Sigma \setminus P) \sqcup Q$$
$$P * Q \stackrel{\text{def}}{=} \{\sigma \mid \sigma = \sigma_1 * \sigma_2,\ \sigma_1 \in P,\ \sigma_2 \in Q \text{ for some } \sigma_1, \sigma_2 \in \Sigma\}$$
$$\top * U = U * \top \stackrel{\text{def}}{=} \top$$
$$[f]U \stackrel{\text{def}}{=} \{\sigma \mid f(\sigma) \sqsubseteq U\}$$

Using these operations a model extends the mapping $[\![\cdot]\!]$ to all formulas: for example, $[\![F_1 \wedge F_2]\!] \stackrel{\text{def}}{=} [\![F_1]\!] \wedge [\![F_2]\!]$. In addition, the semantics of atomic formulas is defined.

$$[\![\mathsf{true}]\!] = \Sigma, \qquad [\![\mathsf{false}]\!] = \emptyset, \qquad [\![\mathsf{emp}]\!] = \{e\}$$

It's important to note that for $[\![F]\!]$ is a predicate, not $\top$, for every formula F.

The mapping $[\![\cdot]\!]$ is extended to all commands as follows.

$$[\![C_1; C_2]\!] = [\![C_1]\!]; [\![C_2]\!]$$
$$[\![C^*]\!] = \bigsqcup_{n=0}^{\infty} [\![C]\!]^n$$
$$[\![C_1 + C_2]\!] = [\![C_1]\!] \sqcup [\![C_2]\!]$$
$$[\![F?]\!](\sigma) = \begin{cases} \{\sigma\}, & \forall \sigma'.\ \sigma \preceq \sigma' \implies \sigma' \models F \\ \emptyset, & \forall \sigma'.\ \sigma \preceq \sigma' \implies \sigma' \not\models F \\ \top, & \text{otherwise} \end{cases}$$
$$(f; g)(\sigma) = \begin{cases} \top, & f(\sigma) = \top \\ \bigsqcup\{g(\sigma') \mid \sigma' \in f(\sigma)\}, & \text{otherwise} \end{cases}$$
$$f^0 = \mathsf{id}_\Sigma$$
$$f^{n+1} = f; f^n$$

The definition of $[\![C]\!]$ is expected possibly with the exception of $[\![F?]\!]$: in regular dynamic logic $[\![F?]\!](\sigma)$ is simply defined as $\{\sigma\}$ if $\sigma \models F$ and $\emptyset$ otherwise [7, Sect. 5.2].

It is proved in [9, Lemma 9] that local actions form a complete lattice, which, together with the way $[\![C]\!]$ is defined, implies that all actions corresponding to commands are local. We consider the case of $F?$, which is stated in [9] but not proved in detail.

Suppose $\sigma_1 \preceq \sigma$ implies $\sigma \models F$ for all σ. Then the same property holds when σ_1 is replaced with $\sigma_1 * \sigma_2$. Therefore, $[\![F?]\!](\sigma_1) = \{\sigma_1\}$, $[\![F?]\!](\sigma_1 * \sigma_2) = \{\sigma_1 * \sigma_2\}$ and $[\![F?]\!](\sigma_1 * \sigma_2) \subseteq [\![F?]\!](\sigma_1) * \{\sigma_2\}$ holds since both sides are equal to $\{\sigma_1 * \sigma_2\}$.

Suppose $\sigma_1 \preceq \sigma$ implies $\sigma \not\models F$ for all σ. Then again the same property holds when σ_1 is replaced with $\sigma_1 * \sigma_2$, so $[\![F?]\!](\sigma_1) = [\![F?]\!](\sigma_1 * \sigma_2) = \emptyset$, from which the desired conclusion follows.

Finally, if none of the cases above is true, then $[\![F?]\!](\sigma_1) = \top = [\![F?]\!](\sigma_1) * \{\sigma_2\}$ by Definition 8, so the property holds since $\top$ is the greatest element.

A natural question asks which formulas F define actions $[\![F?]\!]$ that don't return $\top$, i.e., terminate normally. One such class of formulas does not depend on the heap but uses only local variables, i.e., the store. Since the content of the heap can be loaded into local variables, this does not represent an essential restrictions and corresponds to conditions used in the if operators in popular languages like C and Java. To take advantage of such formulas one has to use predicate dynamic logic rather than propositional dynamic logic considered in this paper.

Soundness of Dynamic Separation Logic

Lemma 12. Let $M = (\Sigma, *, e, [\![\cdot]\!])$ be a model and $f : \Sigma \to P(\Sigma)^\top$ be an action. Then the following conditions are equivalent:

1. f is local, i.e., $f(\sigma_1 * \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\}$ for all $\sigma_1 \# \sigma_2$;
2. $[f]P * Q \sqsubseteq [f](P * Q)$ for all predicates P, Q;
3. $R \sqsubseteq [f]P$ implies $R * Q \sqsubseteq [f](P * Q)$ for all predicates P, Q, R.

Proof. $(1) \implies (2)$. Fix P and Q and suppose $\sigma \models [f]P * Q$. Then $\sigma = \sigma_1 * \sigma_2$ where $\sigma_1 \models [f]P$ and $\sigma_2 \models Q$ for some $\sigma_1, \sigma_2 \in \Sigma$. By definition $f(\sigma_1) \sqsubseteq P$ and $\{\sigma_2\} \sqsubseteq Q$. By locality $f(\sigma_1 * \sigma_2) \sqsubseteq f(\sigma_1) * \{\sigma_2\} \sqsubseteq P * Q$. Here we use mono-

tonicity of $*$ with respect to $\sqsubseteq$, which is checked straightforwardly. Therefore, $\sigma \models [f](P * Q)$.
(2) $\Longrightarrow$ (3). Fix P, Q, R and assume that $R \sqsubseteq [f]P$ and $\sigma \models R * Q$. Then for some σ_1, σ_2 such that $\sigma = \sigma_1 * \sigma_2$ we have $\sigma_1 \models R$, from where $\sigma_1 \models [f]P$, and $\sigma_2 \models Q$, i.e., $\sigma \models [f]P * Q$. By assumption, $\sigma \models [f](P * Q)$.
(3) $\Longrightarrow$ (1). Fix arbitrary σ_1 and σ_2 and assume $\sigma_1 \# \sigma_2$. If $f(\sigma_1) = \top$, then $f(\sigma_1) * \{\sigma_2\} = \top$ and the claim follows. Otherwise, let $P = f(\sigma_1)$, $Q = \{\sigma_2\}$ and $R = [f]P$. Then $\sigma_1 \models [f]P$ and $\sigma_2 \models Q$, that is, $\sigma_1 * \sigma_2 \models [f]P * Q$. By assumption $\sigma_1 * \sigma_2 \models [f](P * Q)$, i.e., $[f](\sigma_1 * \sigma_2) \sqsubseteq P * Q = f(\sigma_1) * \{\sigma_2\}$, as required.

Definition 13. Let $M = (\Sigma, *, e, [\![\cdot]\!])$ be a model. We say that a predicate P is true in M and write $M \models P$ if $P = \Sigma$. A formula F is called true in M (written $M \models F$) if $M \models [\![F]\!]$. A formula is called valid if it is true in every model. An inference rule deriving predicate Q from $P_1, \ldots, P_n$ is sound if $M \models P_1, \ldots, M \models P_n$ implies $M \models Q$ for every M. Similarly, an inference rule deriving formula G from $F_1, \ldots, F_n$ is sound if $M \models F_1, \ldots, M \models F_n$ implies $M \models G$ for every M.

Definition 14. A Hilbert-style deductive system for dynamic separation logic is given by the following axioms and inference rules:

1. Axioms of classical propositional logic;
2. $[C](F \to G) \to ([C]F \to [C]G)$;
3. $[C](F \wedge G) \leftrightarrow ([C]F \wedge [C]G)$;
4. $[C_1 + C_2]F \leftrightarrow [C_1]F \wedge [C_2]F$;
5. $[C_1; C_2]F \leftrightarrow [C_1][C_2]F$;
6. $[F] \wedge [C][C^*]F \leftrightarrow [C^*]F$;
7. $F \wedge [C^*](F \to [C]F) \to [C^*]F$;
8. $[C]F * G \to [C](F * G)$;
9. $\dfrac{F \quad F \to G}{G}$;
10. $\dfrac{F}{[C]F}$;
11. $\dfrac{F * \mathsf{true} \to F \quad \neg F * \mathsf{true} \to \neg F}{[F?]G \leftrightarrow (F \to G)}$.

Theorem 15 (Soundness). The deductive system of Definition 14 is sound, i.e., every derivable formula is valid.

Proof. One has to show that all axioms are valid and all inference rules are sound. Axioms 1–7 are standard axioms of dynamic logic, which are valid in all Kripke models. Axiom 8, called the frame axiom, is valid by Lemma 12 since actions corresponding to all commands are local. Inference rules 9 and 10 are also standard for a modal logic. Finally, dynamic logic has the axiom that is the conclusion of rule 11. The direction $\to$ is valid in our setting as well, but as for the opposite direction, $[\![F?]\!](\sigma)$ may return $\top$, in which case $\sigma \not\models [F?]G$. To prevent $[\![F?]\!](\sigma) = \top$ formula F must be true in all σ' such that $\sigma \preceq \sigma'$ or F must be false in all σ' such that $\sigma \preceq \sigma'$. This is ensured by the assumptions of rule 11. If $\sigma' \models F * \mathsf{true} \to F$ and $\sigma \models F$ for some $\sigma \preceq \sigma'$, then $\sigma' \models F$ as well, and similarly for $\neg F$.

Frame rule (1) can be written in dynamic separation logic as follows.

$$\frac{R \to [C]P}{R * Q \to [C](P * Q)}$$

Points 2 and 3 of Lemma 12 shows that it is sound and can replace the frame axiom.

The additional flexibility of dynamic logic with respect to regular separation logic make several primitive inference rules of the latter derivable. For example, [9] has the rule

$$\frac{\{F_i\}C\{G_i\},\ i = 1, \ldots, n}{\{\bigvee_{i=1}^n F_i\}C\{\bigvee_{i=1}^n G_i\}} \qquad \frac{\{F_i\}C\{G_i\},\ i = 1, \ldots, n}{\{\bigwedge_{i=1}^n F_i\}C\{\bigwedge_{i=1}^n G_i\}}$$

In dynamic logic we have the following facts about derivability: if $\Gamma = \{F_i \to [C]G_i \mid i = 1, \ldots, n\}$, then

$$\Gamma \vdash \bigvee_{i=1}^n F_i \to [C]\Big(\bigvee_{i=1}^n G_i\Big) \quad \text{and} \quad \Gamma \vdash \bigwedge_{i=1}^n F_i \to [C]\Big(\bigwedge_{i=1}^n G_i\Big).$$

Separation Logic in Education

Dynamic and separation logics are excellent topics for introducing students to formal methods and theoretical computer science. Though they require certain mathematical maturity, they are accessible to undergraduate students. In fact, this article uses hardly any concepts that cannot be considered in the introductory courses of discrete mathematics and mathematical logic.
The logic of bunched implications, on which separation logic is built, is an example of a substructural logic, where the structural rules of weakening and contraction

$$\frac{\Gamma \vdash F}{\Gamma, G \vdash F} \qquad \frac{\Gamma, G, G \vdash F}{\Gamma, G \vdash F}$$

are not generally valid. This reveals itself in the fact that the formula $F * F \leftrightarrow F$ is not valid, in contrast to $F \wedge F \leftrightarrow F$. Indeed, $\sigma \models F * F$ means that the state σ can be divided into σ_1 and σ_2, each of which validate F. This is different from F being true on the whole state.

Another prominent substructural logic is linear logic [11, Chap. 9], which is often used to describe resources. However, separation logic is more accessible to new students due to its simple heap (or, more generally, separation algebra) semantics. Even though classical logic can be embedded into linear logic via a translation, the BI logic is simply an extension of classical logic. A student just needs to understand the semantics of separation conjunction and separating implication (the latter is more rarely used in practice).
Thus studying separation logic introduces a student to a world of non-classical logics in a natural way. It demonstrates how a logic can create a model of a computational process and solve a nontrivial problem of tracking pointers.
Hoare logic and separation logic represent a natural entry into the world of formal verification since they can be naturally encoded in a proof assistant such as Coq or Isabelle. This allows formalizing proofs both of properties about the logic themselves (such as soundness and completeness) and of algorithm specifications [12]. As the author of one book on formal methods put it speaking about correctness of programs, “the author and many other researchers today feel that proofs on paper have outlived their usefulness. Instead, the proofs are all found in the parallel world of the accompanying Coq source code” [13]. It turns out that formal verification of even well-known algorithms, such as computing vertex cover or independent set, can uncover incompletenesses in existing proofs and improve the complexity bounds [14].
A great success story in formal verification is the electronic text-

book “Software Foundations” by B.C. Pierce et al., which is widely used around the world. This book in five volumes covers a multitude of topics: propositional and predicate logic, definitions and proofs by induction, Hoare logic, simply typed lambda calculus, functional programming, and so on. All definitions, theorem and proofs in the book are implemented in Coq. The latest volume covers specifying and verifying real-world C programs using separation logic. A similar book based on Isabelle is [15-16]. Another textbook that uses Coq and separation logic is [17]. Dynamic logic is used, for example, in the “Bug Catching” course [18] at Carnegie Mellon University. This course also utilizes a fully automatic system Why3 [19] based on Hoare logic for proving program correctness.

Despite being accessible to students, separation logic is located at the cutting edge of research in computer science [20; 21]. Many software tools have been developed that allow verifying realistic and intricate programs [22]. For example, a significant portion of an industrial, preemptive OS kernel has been verified [23]. Peter O’Hearn, one of the researchers at the origin of separation logic, joined Facebook and developed a static analyzer tool that catches thousands of bugs per month [24].

Using this research may help develop computer science curriculum that takes professional standards seriously and actively uses educational software [25–27]. For these reasons it seems a good idea to teach separation logic in mandatory and optional courses as an excellent introduction to modern computer science.

One personal positive example for the author was supervising a master’s thesis devoted to proving soundness of the frame rule for separation logic with regular programs used in this paper as opposed to while programs in [5]. The project received an excellent grade.

Conclusion and Future Work

We have introduced propositional dynamic separation logic and proved its soundness. We have also argued for including separation logic into computer science curriculum as an excellent introduction to both research problems and practical software verification.

The next obvious step is proving completeness of dynamic separation logic, considering its first-order variant and studying whether the use of dynamic logic creates advantages over regular separation logic that uses Hoare triples in practical program verification. Another important variant of separation logic that can be converted to dynamic flavor is concurrent separation logic [22].

References

[1] Clarke E.M., Wing J.M. Formal methods: state of the art and future directions. *ACM Computing Surveys*. 1996; 28(4):626-643. (In Eng.) DOI: https://doi.org/10.1145/242223.242257

[2] Hoare C.A.R. An axiomatic basis for computer programming. *Communications of the ACM*. 1969; 12(10):576-580. (In Eng.) DOI: https://doi.org/10.1145/363235.363259

[3] Apt K.R., Olderog E.R. Fifty years of Hoare’s logic. *Formal Aspects of Computing*. 2019; 31(6):751-807. (In Eng.) DOI: https://doi.org/10.1007/s00165-019-00501-3

[4] Reynolds J.C. Separation logic: a logic for shared mutable data structures. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* Copenhagen, Denmark; 2002. p. 55-74. (In Eng.) DOI: https://doi.org/10.1109/LICS.2002.1029817

[5] Yang H., O’Hearn P. A Semantic Basis for Local Reasoning. In: Nielsen M., Engberg U. (ed.) Foundations of Software Science and Computation Structures. FoSSaCS 2002. *Lecture Notes in Computer Science*. 2002; 2303:402-416. Springer, Berlin, Heidelberg. (In Eng.) DOI: https://doi.org/10.1007/3-540-45931-6_28

[6] O’Hearn P.W., Pym D.J. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*. 1999; 5(2):215-244. (In Eng.) DOI: https://doi.org/10.2307/421090

[7] Harel D., Kozen D., Tiuryn J. Dynamic Logic. MIT Press; 2000. Available at: https://mitpress.mit.edu/books/dynamic-logic (accessed 02.09.2020). (In Eng.)

[8] Ahrendt W., Beckert B., Bubel R., et al. Deductive Software Verification — The KeY Book. From Theory to Practice. In: Ahrendt W., et al. *Lecture Notes in Computer Science*. 2016; 10001. Springer, Cham. (In Eng.) DOI: https://doi.org/10.1007/978-3-319-49812-6

[9] Calcagno C., O’Hearn P.W., Yang H. Local Action and Abstract Separation Logic. In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007).* Wroclaw, Poland; 2007. p. 366-378. (In Eng.) DOI: https://doi.org/10.1109/LICS.2007.30

[10] Parkinson M., Bornat R., Calcagno C. Variables as Resource in Hoare Logics. In: *21st Annual IEEE Symposium on Logic in Computer Science (LICS’06)*. Seattle, WA, USA; 2006. p. 137-146. (In Eng.) DOI: https://doi.org/10.1109/LICS.2006.52

[11] Troelstra A., Schwichtenberg H. Basic Proof Theory. 2nd ed., Cambridge Tracts in Theoretical Computer Science. Cambridge, Cambridge University Press; 2000. (In Eng.) DOI: https://doi.org/10.1017/CBO9781139168717

[12] Chlipala A. Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. MIT Press; 2013. Available at: https://mitpress.mit.edu/books/certified-programming-dependent-types (accessed 02.09.2020). (In Eng.)

[13] Chlipala A. Proof engineering: implementation challenges in rigorously verified software. In: *Proceedings of the Programming Languages Mentoring Workshop (PLMW ‘15)*. Association for Computing Machinery, New York, NY, USA; 2015:8. (In Eng.) DOI: https://doi.org/10.1145/2792434.2792442

[14] Eßmann R., Nipkow T., Robillard S. Verified Approximation Algorithms. In: Peltier N., Sofronie-Stokkermans V. (ed.) Automated Reasoning. IJCAR 2020. *Lecture Notes in Computer Science*. 2020; 12167:291-306. Springer, Cham. (In Eng.) DOI: https://doi.org/10.1007/978-3-030-51054-1_17

[15] Nipkow T. Term rewriting and beyond – theorem proving in Isabelle. *Formal Aspects of Computing*. 1989; 1(1):320-338. (In Eng.) DOI: https://doi.org/10.1007/BF01887212

[16] Nipkow T., Klein G. Concrete Semantics: With Isabelle/HOL. Springer, Cham; 2014. (In Eng.) DOI: https://doi.org/10.1007/978-3-319-10542-0

[17] Sergey I., Wilcox J.R., Tatlock Z. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*. 2017; 2(POPL):28. (In Eng.) DOI: https://doi.org/10.1145/3158116

[18] Bradley A.R., Manna Z. The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Berlin, Heidelberg; 2007. (In Eng.) DOI: https://doi.org/10.1007/978-3-540-74113-8

[19] Filliâtre J.C., Paskevich A. Why3 – Where Programs Meet Provers. In: Felleisen M., Gardner P. (ed.) Programming Languages and Systems. ESOP 2013. *Lecture Notes in Computer Science*. 2013; 7792:125-128. Springer, Berlin, Heidelberg. (In Eng.) DOI: https://doi.org/10.1007/978-3-642-37036-6_8

[20] Krebbers R., Jung R., Bizjak A., Jourdan J.H., Dreyer D., Birkedal L. The Essence of Higher-Order Concurrent Separation Logic. In: Yang H. (ed.) Programming Languages and Systems. ESOP 2017. *Lecture Notes in Computer Science.* 2017; 10201:696-723. Springer, Berlin, Heidelberg. (In Eng.) DOI: https://doi.org/10.1007/978-3-662-54434-1_26

[21] Jung R., Jourdan J.-H., Krebbers R., Dreyer D. RustBelt: securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*. 2017; 2(POPL):66. (In Eng.) DOI: https://doi.org/10.1145/3158154

[22] Brookes S., O'Hearn P.W. Concurrent separation logic. *ACM SIGLOG News*. 2016; 3(3):47-65. (In Eng.) DOI: https://doi.org/10.1145/2984450.2984457

[23] Xu F., Fu M., Feng X., Zhang X., Zhang H., Li Z. A Practical Verification Framework for Preemptive OS Kernels. In: Chaudhuri S., Farzan A. (ed.) Computer Aided Verification. CAV 2016. *Lecture Notes in Computer Science*. 2016; 9780:59-79. Springer, Cham. (In Eng.) DOI: https://doi.org/10.1007/978-3-319-41540-6_4

[24] O'Hearn P. Separation Logic. *Communications of the ACM*. 2019; 62(2):86-95. (In Eng.) DOI: https://doi.org/10.1145/3211968

[25] Zakharova I., Kuzenkov O. The Experience of Updating the Educational Standards of Higher Education in the Field of ICT. *Sovremennye informacionnye tehnologii i IT-obrazovanie* = Modern Information Technologies and IT-Education. 2017; 13(4):46-57. (In Russ., abstract in Eng.) DOI: https://doi.org/10.25559/SITITO.2017.4.510

[26] Zakharova I., Kuzenkov O. Mathematical Programs Modernization Based on Russian and International Standards. *Sovremennye informacionnye tehnologii i IT-obrazovanie* = Modern Information Technologies and IT-Education. 2018; 14(1):233-244. (In Eng., abstract in Russ.) DOI: https://doi.org/10.25559/SITITO.14.201801.233-244

[27] Kuzenkov O., Kuzenkova G., Kiseleva T. The use of electronic teaching tools in the modernization of the course "Mathematical modeling of selection processes". *Obrazovatel'nye tehnologii i obshhestvo* = Educational Technology & Society. 2018; 21(1):435-448. Available at: https://elibrary.ru/item.asp?id=32253185 (accessed 02.09.2020). (In Russ., abstract in Eng.)

Submitted 02.09.2020; approved after reviewing 14.10.2020; accepted for publication 10.11.2020.
Поступила 02.09.2020; одобрена после рецензирования 14.10.2020; принята к публикации 10.11.2020.

About the author:

Evgeny M. Makarov, Senior Instructor of the Department of Algebra, Geometry and Discrete Mathematics, Institute of Information Technology, Mathematics and Mechanics, Lobachevsky State University of Nizhny Novgorod (23 Gagarin Av., Nizhny Novgorod 603950, Russian Federation), Ph.D. (Computer Science), ORCID: http://orcid.org/0000-0003-0399-0946, evgeny.makarov@itmm.unn.ru

The author has read and approved the final manuscript.

Об авторе:

Макаров Евгений Маратович, старший преподаватель кафедры алгебры, геометрии и дискретной математики, Институт информационных технологий, математики и механики, ФГАОУ ВО «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского» (603022, Российская Федерация, г. Нижний Новгород, пр. Гагарина, д. 23), ORCID: http://orcid.org/0000-0003-0399-0946, evgeny.makarov@itmm.unn.ru

Автор прочитал и одобрил окончательный вариант рукописи.